Advanced Component Models

Christian Perez
LIP/INRIA
2010-2011

Content

- Algorithmic skeletons
  - Master-worker
  - MapReduce
  - STKM
- Generic Components
- “Classical” Parallelism
  - Data sharing composition
  - NxM Components
  - Collective communications
- Connectors
  - “Classical” connector
  - “Open” connections
- Conclusion
Algorithmic Skeletons

Algorithmic skeletons

compute in (int a) out (float b)
$ sequential code $
end

pipe p in (int a) out (float b)
p1 in (a) out (float b1)
p2 in (b1) out (int b2)
p3 in (b2) out (b)
end pipe

farm f in (int af) out(int bf)
w in (af) out(bf)
end farm
pipe pp in (float a) out(float b)
pp1 in (b) out (int b1)
f in (b1) out (b2)
pp3 in (b2) out (b)
end pipe
Algorithmic skeletons [M. Cole ‘89]

- Predefined patterns for parallel programming
  - Stream parallel
    - Pipeline, farm, if, while, etc
  - Data parallel
    - Fork, divide-and-conquer, Map (independent forAll), reduce, …

- Structured programming
  - Simplicity
  - Correctness of programs

- Hide the complexity of parallelism management
  - Creation of processes, data distribution, ..

- Behavioral skeletons add advanced management for adaptation

Algorithmic Skeletons

Master-Worker Relationships
Master-worker paradigm

- Multiple independent computations (boucle ~ForAll)
- Dedicated environments/API
  - GridRPC: DIET, NetSolve, Ninf-G, …
  - Desktop Grid: BOINC, XtremWEB, …

Characteristics of master-worker environments

- Advanced request transfer policies
- Transparent management of non-functional concerns

- Dedicated APIs
- Limited programming paradigms
Assembling a master-worker application in classical component models

- Non-functional concerns
- Resources dependencies

(A) \( m \rightarrow wi \)

(B) \( m \rightarrow wi \)

(C) \( m \rightarrow wi \)

Abstract assembly

Composition of components with collections

\( \approx \)

Composition of components
Collection of components

Definition of a collection

Collection at execution

Server exposed port

Instantiation

Instantiation

Request transfer patterns

Simple Component base pattern

Hierarchical Component based pattern

DIET pattern
Overview of the proposal

**Designer view**
- **master**
- **worker**

**System/platform view**
- **execution resources**
- **# workers + pattern selection**
- **list of request transfer patterns**
  1. Random
  2. Round-Robin
  3. NetSolve
  4. Diet

**Algorithmic Skeletons**
- **MapReduce**
Motivation: Large Scale Data Processing

- Want to process lots of data (> 1 TB)
- Want to parallelize across hundreds/thousands of CPUs
- … Want to make this easy

MapReduce

- Automatic parallelization & distribution
- Fault-tolerant
- Provides status and monitoring tools
- Clean abstraction for programmers
Programming Model

- Borrows from functional programming
- Users implement interface of two functions:
  - `map (in_key, in_value) -> (out_key, intermediate_value) list`
  - `reduce (out_key, intermediate_value list) -> out_value list`

Distributed Grep

Very big data → grep → matches → cat → All matches

```
<table>
<thead>
<tr>
<th>Split data</th>
<th>grep</th>
<th>matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split data</td>
<td>grep</td>
<td>matches</td>
</tr>
<tr>
<td>Split data</td>
<td>grep</td>
<td>matches</td>
</tr>
</tbody>
</table>
```
Distributed Word Count

Very big data → Split data → count → count → count → ... → merge → merged count

Map Reduce

- **Map:**
  - Accepts *input* key/value pair
  - Emits *intermediate* key/value pair

- **Reduce:**
  - Accepts *intermediate* key/value* pair
  - Emits *output* key/value pair

Very big data → MAP → Partitioning Function → REDUCE → Result
Partitioning Function (1/2)

<table>
<thead>
<tr>
<th>Input</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
</tbody>
</table>

Intermediate:

| k1:v | k1:v | k2:v | k1:v | k3:v | k4:v | k5:v |

Group by Key:

Grouped:

| k1:v | k1:v | k2:v | k3:v | k4:v | k5:v |

Output:

Partitioning Function (2/2)

- **Default**: \( \text{hash(key)} \mod R \)
- **Guarantee**:  
  - Relatively well-balanced partitions  
  - Ordering guarantee within partition
- **Distributed Sort**  
  - Map:
    
    ```java
    emit(key, value)
    ```
  - Reduce (with \( R=1 \)):
    
    ```java
    emit(key, value)
    ```
MapReduce

- Distributed Grep
  - Map:
    \[
    \text{if } \text{match}(\text{value}, \text{pattern}) \text{ emit}(\text{value}, 1)
    \]
  - Reduce:
    \[
    \text{emit}(\text{key}, \text{sum}(\text{value}^*))
    \]

- Distributed Word Count
  - Map:
    \[
    \text{for all } w \text{ in } \text{value} \text{ do emit}(w, 1)
    \]
  - Reduce:
    \[
    \text{emit}(\text{key}, \text{sum}(\text{value}^*))
    \]
Limitations of existing component models

- Assembly models close to the computing resources
  - Behavior hidden in the assembly
  - "Over-consumption" of resources
  - Simple spatial relations
    - Resource dependencies
    - Complex design
      - Parallel paradigms (e.g. master-worker)

Workflow models

Algorithmic skeleton models

Objectives

- Simplifying programming parallel parts of an application
- Offering a similar level of abstraction as in skeleton models
- Portability on different execution resources
  - Code reuse
  - Efficiency
Overview of STKM

- Assembly model
  - STCM assembly + skeleton constructs
  - An STKM skeleton is a composite with a predefined behavior
- Parameterization
  - Wrapping components
- Usage in spatial and temporal dimension
  - Port cardinality principle (temporal dimension)

Component wrapping and port cardinality principle

- A skeleton element is a wrapped component
- Port cardinality
STKM: Assembly model

```plaintext
component Example {
  ... Step1 and Step3 components...

farm Step2 {
  inputSkel double inS2;
  outputSkel string outS2;

  worker sequential w {
    inputSkel double inW;
    outputSkel string outW;

    component Worker {
      streamIn double inW;
      streamOut string outW;

      connect outW to Worker.outW;
      connect Worker.inW to inW;
    }
  }

  instances: Step1 step1; Step2 step2; Step3 step3;
  ... Connexions step1 <=> step2 <=> step3 ...

sequence ApplMain {
  exec step1; exec step2; exec step3;
}
```
Motivating example: Overview

- Goal: Generating Mandelbrot set pictures
  - Embarrassingly parallel

- Parallel hardware resources:
  - Ex: Quad-core computer

- Programming pattern: Task-farm skeleton
  - 1 data stream
  - n parallel workers

\[ C = (x, y) \]
\[ Z_{n+1} = Z_n^2 + C \]
- Bounded \( \rightarrow \) black
- Unbounded \( \rightarrow \) blue
Motivating Example: A component based implementation

Motivating example: Limitations to reuse

- Hard-coded in the composite
  - Transformation algorithm
  - Manipulated data-types
  - Number of workers
Motivating Example: A generic farm

MandelbrotFarm
- Mandel Worker
  - Coord Disp
  - Pixel
  - Coord Disp
  - Pixel

MandelbrotFarm<W>
- W
  - Coord Disp
  - Pixel
  - Coord Disp
  - Pixel
  - Coord Disp
  - Pixel
Motivating Example: A generic farm

MandelbrotFarm\langle W, I, O \rangle

\[ \text{Disp} \langle I \rangle \quad \text{Coll} \langle O \rangle \]

Motivating Example: A generic farm

MandelbrotFarm\langle W, I, O, N \rangle

\[ \text{Disp} \langle I \rangle \quad \text{Coll} \langle O \rangle \]

Genericity

\[ \text{N Times} \]
Genericity study: Concepts definitions (1)

```
public class GenClass<T> {
    T member;
    ...
}
```

Java

```
template<typename T>
T genFunc () {
    T locvar;
    ...
    Return locvar;
}
```

C++

**Generic artifacts:**
- Accept 2nd order **parameters**
- Use the parameters in their implementation / body

Genericity study: Concepts definitions (2)

```
GenClass<String> l = new GenClass<String>();
```

Java

```
int i = genfunc<int>();
```

C++

**Specializations:**
- Use of generic artifacts
- Arguments bind parameters to a value
Genericity study: Concepts definitions (3)

Template<>
void* genFunc<void*>(...) {
    ...
}

C++

Explicit Specializations:
- Distinct implementation for a range of specializations
- When some constraints on the parameters are fulfilled

GenCmp<C>

When C.p instanceof G

GenCmp<C>

Toward a generic component model

- Generic concepts
  - Component types
  - Port types

- Concepts as parameters
  - Component types
  - Port interfaces
  - Data types
  - Data values

- Instantiation of parameter types

- Meta-programming
  - Ex: N times replication

- Reuse existing component models
  - Extension of existing models
Genericity study: Type erasure vs. Specialized compilation

- Type erasure: (ex. Java)
  - Type parameters used for checking
  - Only one code compiled, manipulates Object ptrs
  + Compiled code is smaller
  - Limited use of parameter types (no instantiation, limited access to methods, …)

- Specialized compilation (ex. C++)
  - Type parameters replaced in the code
  - One code compiled / specialization
  - No dynamic instantiation of specializations
  + Explicit specializations & template metaprogramming

GenericSCA: Introduced features

- Concepts made generic:
  - Composite component implementations
  - Java component implementations
  - Java port interfaces

- Concepts that can be parameters
  - Component implementations
  - Port interfaces
  - Data-types
  - (Data-values): properties are already part of SCA
Task Farm in GenericSCA: Modeling the Farm

- Six parameters:
  - \( I, O \): type of input & output data
  - \( D, W, C \): Dispatcher, Worker & Collector implementations
  - \( N \): number of Workers
- Default values for \( D \) & \( C \)
- Flow simulated by a DataPush\(<T>\) interface
  - Single method: void push\((T \ data)\);

Task Farm in GenericSCA: The Replication Component

- Recursive implementation
  - When \( R = 1 \)
    - 1 C instance only
  - When \( R > 1 \)
    - 1 C instance
    - 1 Replication instance with \( R \) decreased by 1
Task Farm in GenericSCA: Transformation Example

Farm\(<I=\text{PictRect}, O=\text{ComputedPictRect}, W=\text{Mandel}, N=3>\)

- Replication worker
  - \(T=\text{PictRect}\)
  - \(I=\text{PictRect}, O=\text{ComputedPictRect}, R=3, C=\text{Mandel}\)

- Replication worker
  - \(T=\text{ComputedPictRect}\)

- RRDisp
dispatcher
  - \(T=\text{PictRect}\)

- RRColl
collector
  - \(T=\text{ComputedPictRect}\)
Task Farm in GenericSCA: Transformation Example

Replication\(<I=\text{PictRect}, O=\text{ComputedPictRect}, C=\text{Mandel}, R=3>\)

head

id=R-1

tail

I=I, O=O, C=C, R=R-1

Task Farm in GenericSCA: Transformation Example

Replication\(<I=\text{PictRect}, O=\text{ComputedPictRect}, C=\text{Mandel}, R=3>\)

head

id=2

tail

I=\text{PictRect}, O=\text{ComputedPictRect}, C=\text{Mandel}, R=2
Task Farm in GenericSCA: Transformation Example

- Replication\(<I=\text{PictRect}, O=\text{ComputedPictRect}, C=\text{Mandel}, R=3>\)
- Replication\(<I=\text{PictRect}, O=\text{ComputedPictRect}, C=\text{Mandel}, R=2>\)
- Replication\(<I=\text{PictRect}, O=\text{ComputedPictRect}, C=\text{Mandel}, R=1>\)

Task Farm in GenericSCA: Transformation Example

- Replication\(<I=\text{PictRect}, O=\text{ComputedPictRect}, C=\text{Mandel}, R=3>\)
- Replication\(<I=\text{PictRect}, O=\text{ComputedPictRect}, C=\text{Mandel}, R=2>\)
- Replication\(<I=\text{..., O=..., C=..., R=1}>\)

Task Farm in GenericSCA: Transformation Example

- Replication\(<I=\text{PictRect}, O=\text{ComputedPictRect}, C=\text{Mandel}, R=3>\)
- Replication\(<I=\text{PictRect}, O=\text{ComputedPictRect}, C=\text{Mandel}, R=2>\)
- Replication\(<I=\text{..., O=..., C=..., R=1}>\)
“Classical” Parallelism in Component Models

Data sharing
NxM
Collective Communications

Data-sharing Composition
Data sharing

- Multiple concurrent accesses to a data
- Localization and concurrent accesses management
  - Intra-machine: OS
  - Intra-cluster: Distributed shared memory (DSM)
  - Intra-grid: sharing data service (JuxMem/PARIS)

Limits with data sharing in component models

- Ports: active communication operation
  - Data must be part of a message

- Centralized approach
  - Bottleneck for the performance
  - Single point of failure

- Distributed approach
  - Explicit management of data replication/migration by components
  - Functional code mixed with data management code
Overview of the Model

- Principle: transparent access to a shared data

\textit{accesses} float port \textit{shares} float port

\begin{verbatim}
interface SharesPort {
    float* allocate_space(in long size);
    void free_space();
    float* get_pointer();
    long get_size();
    void acquire();
    void release();
    void acquire_release();
}
\end{verbatim}

Example of data sharing ports

```java
class ComImpl {
    Services srv;
    AccessPort myPort;
    ...
    void computeSum(){
        myPort = srv.getPort("myPort");
        myPort.acquireR();
        ptr = myPort.getPointer();
        size = myPort.get_size();
        for (i = 0; i < size; i++)
            sum += ptr[i];
        myPort.release();
    }
}
```

```java
interface ExtendedServices : Services {
    ...
}
```

```java
interface AccessPort : Port {
    opaque get_pointer();
    long get_size();
    void acquire();
    void acquireR();
    void release();
}
```

```java
interface SharesPort : AccessPort {
    void associate (in opaque ptr, in long size);
    void disassociate();
}
```

Selected depending on resources and comp. placement

- OS, DSM, JuxMem
“Classical” Parallelism in Component Models

MxN Communications

Application in hydrogeology: Saltwater intrusion

- Coupled physical models
- One model = one software
- Saltwater intrusion
  - Flow / transport
- Reactive transport
  - Transport / chemistry
- Hydrogrid project, supported by the French ACI-GRID

flow: velocity and pressure function of the density
Density function of salt concentration
Salt transport: by convection (velocity) and diffusion
Numerical coupling in saltwater intrusion

Components and communications of PCSI
Components and interfaces of PCSI

Limits to MxN code communications

Flat programming model (à la MPI)

Bridge based solution
SPMD Components

What the application designer should see…

… and how it must be implemented!

Distributed Component Model

// Emitter Code
o.factorize(m);

// Receiver Code
void serv::factorize(const Matrix mat) {
    ...
}
SPMD Parallel Component Model (1)

// Emitter Code
void factorize(const DMatrix mat) {
    // MPI_Bcast(...) ...
}

Parallel Distributed Component

// Receiver Code
void factoriser(const Matrice mat) {
    ...
}

Distributed component (a process)

Parallel Distributed Component

Data Redistribution

SPMD Parallel Component Model (2)

// SPMD Emitter Code
o.factorize(m);

Parallel Caller

// Receiver Code
void factoriser(const Matrice mat) {
    ...
}

Receiver

Distributed component (a process)

Parallel Distributed Component
**SPMD Parallel Component Model (3)**

```
// SPMD Emitter Code
o->factorize(m);

// SPMD Receiver Code
void factorizer(const Matrix mat)
{ .... MPI_Bcast(...) ... ;}
```

**GridCCM Component**

```
interface IExample
{
    void factorise(in Matrix mat);
};
```

```
component CoPa
{
    provides IExample to_client;
    uses IFaces2 to_server;
};
```

**XML**

```
Component: CoPa
Port: to_client
Name: IExample.factorise
Type: Parallel
Argument1: Basic_BC[*, bloc]
ReturnArgument: noReduction
```
Components for code coupling: SPMD paradigm in GridCCM

- SPMD component
  - Parallelism is a non-functional property of a component
    - It is an implementation issue
  - Collection of sequential components
    - SPMD execution model
  - Support of distributed arguments
    - API for data redistribution
    - API for communication scheduling w.r.t. network properties
  - Support of parallel exceptions

“Classical” Parallelism in Component Models

Parallelism in Common Component Architecture
CCA Supports Parallelism by “Staying Out of the Way” of it

- Single component multiple data (SCMD) model is component analog of widely used SPMD model
  - Each process loaded with the same set of components wired the same way
  - Different components in same process “talk to each” other via ports and the framework
  - Same component in different processes talk to each other through their favorite communications layer (i.e. MPI, PVM, GA)

- Components: Blue, Green, Red
- Framework: Gray

Any parallel programming environments that can be mixed outside of CCA can be mixed inside

---

“Multiple-Component Multiple-Data” Applications in CCA

- Simulation composed of multiple SCMD sub-tasks

- Usage Scenarios:
  - Model coupling (e.g. Atmosphere/Ocean)
  - General multi-physics applications
  - Software licensing issues
    - i.e. limited number of instances

- Approaches
  - Run single parallel framework
    - Driver component that partitions processes and builds rest of application as appropriate (through BuilderService)
  - Run multiple parallel frameworks
    - Link through specialized communications components
    - Link as components (through AbstractFramework service)
Components only on process group B

MCMD Within A Single Framework

Framework
Application driver & MCMD support component
Components on all processes
Components only on process group A
Components only on process group B

P0 P1 P2 P3

Group A Group B

“Classical” Parallelism in Component Models

Collective Communications
Parallelism in component models

- Using message passing libraries (ex. MPI) inside components
- Using parallel ports (NxM) between components
- No collective communications at higher level
- Two communication models to handle

Collective communications

- Most effective algorithm depend on the resources
- On hierarchic resources
  - hierarchic algorithms

Goals: Overview

- Collective communications between components
  - Efficient
  - Transparent
  - Fits in component model

Goals: From the user point of view

- Collective communications are a service
  - Provided by a component

- Communications groups are a way to connect component instances
  - Described in the assembly
Goals: From the developer point of view

- Efficiency
  - Decentralized implementation

Goals: From the developer point of view

- Efficiency
  - Decentralized implementation
- Communications between processes
  - Alltoall connection
Goals: From the developer point of view

- **Efficiency**
  - Decentralized implementation

- **Communications between processes**
  - Alltoall connection

- **Hierarchical resources & algorithm**
  - Hierarchical assembly
Collective Communications

A component model with replicating component

Generic model: Assumptions

- A component model with
  - Components with multiple implementations
  - Primitive & composite component
  - Replicating component
  - An ADL to describe the assembly

- A resource model

- An algorithm to expand the assembly
Generic model: Replicating component & Resource model

An algorithm to expand the assembly

Collective Communications

Usage example
Usage example: hierarchic broadcast

Broadcast Provider

C1
User P1
User P2

C2
User P3
User P4

Usage example: hierarchic broadcast

Broadcast Provider

C1
User P1
User P2

C2
User P3
User P4

Broadcast Provider

MatsudaC Replicate

1 / cluster
Usage example: hierarchic broadcast

MatsudaCReplicate

Scatter Provider

Allgather Provider

Matsuda Broadcast

P1

P2

P3

P4
Usage example: hierarchic broadcast

![Diagram of hierarchic broadcast]

Usage example: hierarchic broadcast

![Diagram of hierarchic broadcast]
Usage example: hierarchic broadcast

Usage example: hierarchic broadcast
Preliminary experiments: The underlying resources

- **Software**
  - Projection on CCM
    - Homemade CCM → CORBA compiler
    - OmniORB 4.1
  - Handmade ADL transformation
  - Comparison with GridMPI

- **Hardware**
  - Grid5000, French experimental platform
  - 2 clusters: Rennes & Sophia Antipolis
  - Latences:
    - Inside cluster: 50µs
    - Between clusters: 10ms
  - Bandwith:
    - Node network card: 1Gb/s
    - Backbone: 10Gb/s

Preliminary experiments: Performances & analysis

Broadcast duration on 2 clusters of 8 nodes

- CCM implementation uses Matsuda algorithm
- GridMPI does not

- CCM does synchronous calls
- GridMPI is asynchronous
Connector-based Composition

Notion of connector

- Introduced in ADL
  - Architecture Description Language
- First class entities
  - List of named roles, with or without cardinality constraints
  - Roles are fulfilled by components’ ports
- Instantiated by connection
- Implemented by generator
- Example
  - Connector mpi<role participant>
  - Connector UP<role user
    role provider>
  - Connector consensus<…>
High Level Component Model

Hierarchy, Genericity, Template Meta-Programming & Connectors

- Major concepts
  - Hierarchical model
  - Generic model
    - Support meta-programming (template à la C++)
  - Connector based
    - Primitive and composite
  - Currently static

- HLCMi: an implementation of HLCM
  - Model-transformation based
  - Already implemented connectors
    - Use/Provide, Shared Data, Collective Communications, "MxN" RMI, Irregular Mesh
Connectors

- Without connectors
  - Direct connection between ports
  - Limitation to 1:1 connection

- With connectors
  - Connectors reify connections
    - A name
    - A set of roles
  - Any number of roles
  - Can be 1st class entities
    - Implemented by the user

Connector implementations

- Intrinsically generic
  - Types of roles fulfillment ↔ parameters for the implementation
- 1 connector ↔ multiple implementations
  - For distinct placement on hardware resources
- Two possible kinds
  - Primitive connectors
    - Directly supported by the model
  - Composite connectors
    - An assembly

When PT subtype of UT and user.host = provider.host
Example of More Complex Interactions as Connectors

- Shared data between components
  - One single role
  - Multiple fulfillments

- Parallel method calls
  - Provides the redistribution
  - An example
    - 2x2 Matrix multiplication
    - 2 roles for users (top/bottom)
    - 2 roles for providers (right/left)

Notion of Open Connections

- Components expose “open connections”
  - Some roles fulfilled
  - Some roles left “open”

- Interactions are defined by “merging” connections
  - Union of the role fulfillments
  - A single logical connection
Expressing Parallel Matrix Multiplication with HLCM

Results in

What implementation to use for this connection?

Connection Implementation: a Planning Choice

Single host distribution

Multiple hosts distribution
HLCM/CCM/MxN vs PaCO++

Conclusion

- From « simple » to « complex » composition operators
- Need of models with open composition support
  - Component, connector, hierarchy, genericity, etc.
- Need of models/algorithms to derive actual implementation from an abstract declaration
- Need of models/algorithms to support dynamicity
  - Adaptability: reaction to environment modifications
  - « workflow »: reaction to programmed modifications